The next phase to implement in your compiler is intermediate code generation. The code will be three-address code represented as quadruples (very much as described in the textbook).

**Intermediate Code**

The three-address code language consists of items of the following forms:

- \( x := y \) binop \( z \) where binop is one of: +, -, *, /, and, or, <,<=, =,=>,>. The quadruple representation of this command is given by \([Op \ x \ y \ z]\) where \(Op \in \{\text{PLUS, MINUS, TIMES, DIV, AND, OR, LT, LE, EQ, GE, GT, NE}\}\).

- \( x := \text{unop} \ y \) where unop is one of: -, not. The quadruple representation is given by \([Op \ x \ y \ z]\) where \(Op \in \{\text{NEG, NOT}\}\).

- \( x := y \) The quadruple representation is given by \(\text{ASSIGN} \ x \ y \ -\).

- \( L: \) where \(L\) is a label. The quadruple representation is given by \(\text{LABEL} \ L \ -\ -\).

- \( \text{goto} \ L \) The quadruple representation is given by \(\text{GOTO} \ L \ -\ -\).

- \( \text{if} \ x \ \text{goto} \ L \) This instruction jumps to \(L\) if \(x\) is non-zero. The quadruple representation is given by \(\text{IF} \ L \ x \ -\).

- \( \text{param} \ x \) precedes call operation, passing a parameter. The quadruple representation is given by \(\text{PAR} \ x \ -\ -\). NOTE: these operations should be issued in left-to-right order. The next phase of your compiler, the assembly-code generator, may decide to generate "push" operations in reverse order, but that is machine dependent.

- \( \text{call} \ p,n \) procedure call with \(n \geq 0\) parameters, no return value. The quadruple representation is given by \(\text{CALL} \ p \ n\).

- \( x := \text{fcall} \ f,n \) function call with \(n \geq 0\) parameters, \(x\) is assigned the value returned by \(f\). The quadruple representation is given by \(\text{FCALL} \ x \ f \ n\).

- \( \text{return} \) procedure return. The quadruple representation is given by \(\text{RET} \ -\ -\ -\).

- \( \text{freturn} \ x \) function return, returning \(x\). The quadruple representation is given by \(\text{FRET} \ x \ -\).
x := y[i] \quad In this case, y[i] refers to the (i + 1)'st element of array y, and is independent of the size of the elements. Assembly-code generation, the next phase, will convert this to a size-dependent array reference. The quadruple representation is given by \[ \text{LDAR } x \ y \ i \].

x[i] := y \quad Array store. The quadruple representation is given by \[ \text{STAR } x \ i \ y \].

Note that each of the variables x, y, and z, can correspond to either variables in the source program (in particular, local variables of a procedure, formal parameters of a procedure, or global variables in the program), or temporary variables created by the intermediate code generator. The right-hand side operands can also be constants, in which case, we represent them as “=5” or “=string”.

The code produced out of the source statement

\[
\text{proc}(x, y)
\]

is

\[
\text{param } x; \quad \text{param } y; \quad \text{call proc,2}
\]

Assume that \( x \) has the type \( \text{array}[3..6] \text{ of integer} \), then the source statement \( y := x[5] \) produces the intermediate code \( y := x[2] \).

\section*{Example}

Consider the following function “fact”:

\[
\text{function fact}(x : \text{ integer}) : \text{ integer};
\quad \text{var } u : \text{ integer};
\quad \text{begin}
\quad \quad \text{if } x = 0 \quad \text{then } u := 1
\quad \quad \text{else } u := \text{fact}(x - 1) \times x;
\quad \quad \text{fact} := u
\quad \text{end } \{ \text{fact} \}
\]

The following table contains the intermediate code produced for this function, where on the left we present the C-like representation of the code and, on the left, the quadruple presentation of the same code.
The quadruple table refers to the symbol table which is given by

<table>
<thead>
<tr>
<th>Entry No.</th>
<th>Name</th>
<th>Kind</th>
<th>Type</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>fact</td>
<td>function</td>
<td>integer</td>
<td>global</td>
</tr>
<tr>
<td>2.</td>
<td>x</td>
<td>parameter</td>
<td>integer</td>
<td>fact</td>
</tr>
<tr>
<td>3.</td>
<td>u</td>
<td>variable</td>
<td>integer</td>
<td>fact</td>
</tr>
<tr>
<td>4.</td>
<td>T1</td>
<td>temporary</td>
<td>integer</td>
<td>fact</td>
</tr>
<tr>
<td>5.</td>
<td>T2</td>
<td>temporary</td>
<td>integer</td>
<td>fact</td>
</tr>
<tr>
<td>6.</td>
<td>L1</td>
<td>label</td>
<td>—</td>
<td>fact</td>
</tr>
<tr>
<td>7.</td>
<td>L2</td>
<td>label</td>
<td>—</td>
<td>fact</td>
</tr>
</tbody>
</table>

and to the AST which contains the following fragment:

Note that the code generation process generates temporary variables (such as T1 and T2) and labels (L1, L2) which should be added to the symbol table. Also note that the boolean operation, such as “x=0” generate boolean values which we represent as 1 for true and 0 for false. The first two quadruples place in T2 a value which is 1 iff x is unequal to 0.

**Accessing Structured Variables**

The only intermediate code operations that enable us to access components of a structured type are the indexed references $x[i]$. In case $x$ has a structured type which is not a simple array, we have to compute in $i$ a more complicated index.

We will illustrate the necessary operations on the following example:
type
  foo = array[1..10] of
     record
       x : array[-2..2] of integer;
     end;
var
  i,j,k,num : integer;
  all : foo

To explain the handling of this declaration, we observe that it is equivalent to the following declarations:

type
  foo = array[1..10] of t_1;
  t_1 = record x: t_2; y: t_3 end;
  t_2 = array[-2..2] of integer;
  t_3 = array[1..5] of t_4;
  t_4 = array[4..6] of integer;
var
  i,j,k,num : integer;
  all : foo

where t_1, t_2, t_3, t_4, are anonymous types introduced in order to facilitate the processing. For each of the included types, we compute its size. This is computed as follows:

- If $t$ is a basic type, i.e., integer, string, boolean, then $size(t) = 1$. Note that the size of a string’s entry is also 1, because a standard string entry is a pointer to the actual string which is stores elsewhere.

- If $t$ is an array defined by “$t = \text{array}[c_1..c_2] \text{ of } t_1$”, then $size(t) = (c_2 - c_1 + 1) \times size(t_1)$.

- If $t$ is an record defined by “$t = \text{record } x_1 : t_1; \cdots; x_k : t_k \text{ end}$”, then $size(t) = size(t_1) + \cdots + size(t_k)$.

In addition, for each field designator (such as $x$ or $y$ in the declaration above) we compute a base offset which measures the distance from the beginning of the record. Thus, if type $t$ is a record defined by

$$t = \text{record } x_1 : t_1; \cdots; x_k : t_k \text{ end}$$

then, for each $i = 1, \ldots, k$, we set

$$base(x_i) = size(t_1) + \cdots + size(t_{i-1})$$

Following is a symbol table which contains the types and variables contained in the example declaration. The table contains the additional attributes size and base.
Having computed the necessary attributes for all types and field designators, we can now issue a code for any compound reference, such as \( \text{all}[i].y[j][k] \).

Recall that a compound reference has the general form \( A S_1 \cdots S_k \), where \( A \) is a variable and each \( S_i \) is a selector which is either of the form \( [i] \) or \( .d \), where \( d \) is a field designator. We provide an inductive definition of an expression \( \text{offset}(A S_1 \cdots S_i) \) for each \( i = 1, \ldots, k \).

- \( \text{offset}(A) = 0 \).

- Consider the case that \( S_i = [j] \). In that case, the type of \( A S_1 \cdots S_{i-1} \) must be \( \text{"array\}[c_1\.c_2]\text{ of } t" \). We therefore define

\[
\text{offset}(A S_1 \cdots S_i) = \text{offset}(A S_1 \cdots S_{i-1}) + (j - c_1) \times \text{size}(t)
\]

- Next, consider the case that \( S_i = .d \). In that case, the type of \( A S_1 \cdots S_{i-1} \) must be a record \( t \), where \( d \) is a field designator in \( t \). We therefore define

\[
\text{offset}(A S_1 \cdots S_i) = \text{offset}(A S_1 \cdots S_{i-1}) + \text{base}(d)
\]

Let us apply this definition in order to compute \( \text{offset}(\text{all}[i].y[j][k]) \).

\[
\begin{align*}
\text{offset}(\text{all}) &= 0 \\
\text{offset}(\text{all}[i]) &= 0 + 20(i - 1) = 20i - 20 \\
\text{offset}(\text{all}[i],y) &= 20i - 20 + 5 = 20i - 15 \\
\text{offset}(\text{all}[i],y[j]) &= 20i - 15 + 3(j - 1) = 20i + 3j - 18 \\
\text{offset}(\text{all}[i],y[j][k]) &= 20i + 3j - 18 + (k - 4) = 20i + 3j + k - 22
\end{align*}
\]

Thus, the code that should be issued for the source statement

\[
\text{num} := \text{all}[i].y[j][k]
\]

is given by:

\[
\begin{align*}
T1 &:= i - 1 \\
T1 &:= 20 \ast T1 \\
T1 &:= T1 + 5 \\
T2 &:= j - 1 \\
T2 &:= T2 \ast 3 \\
T1 &:= T1 + T2 \\
T2 &:= k - 4 \\
T1 &:= T1 + T2 \\
\text{num} &:= \text{all}[T1]
\end{align*}
\]
Representation of quadruples

Quadruples can be represented by a record (or C++/Java object) with four fields: op, target, arg1, and arg2. The target, arg1, and arg2 fields might refer to variables in the source program or to temporary variables created by the intermediate code generator. If the former, then those fields should point to the node in the symbol table containing the type information, etc. that your parser and type checker created. Otherwise, as your intermediate code generator generates a temporary variable, it should create a node with the necessary type information, which the field in the quadruple will point to.

Each procedure in the program can be represented by an array of quadruples. This array should be contained in a record or object that also points to the node giving type and formal parameter information about the procedure.

Deliverables

For each processed program, print the symbol table and AST and also the intermediate generated code in the Micro-C format as introduced above.

Run your compiler on the test programs test0009.pas — test0025.pas. Submit your program with the output produced for these test programs.